

# Simplification of integrity constraints with aggregates and arithmetic built-ins

Davide Martinenghi

Roskilde University, Computer Science Dept.  
P.O.Box 260, DK-4000 Roskilde, Denmark  
E-mail: [dm@ruc.dk](mailto:dm@ruc.dk)

**Abstract.** Both aggregates and arithmetic built-ins are widely used in current database query languages: Aggregates are second-order constructs such as `COUNT` and `SUM` of SQL; arithmetic built-ins include relational and other mathematical operators that apply to numbers, such as  $\leq$  and  $+$ . These features are also of interest in the context of database integrity constraints: correct and efficient integrity checking is crucial, as, without any guarantee of data consistency, the answers to queries cannot be trusted. In this paper we propose a method of practical relevance that can be used to derive, at database design time, simplified versions of such integrity constraints that can be tested before the execution of any update. In this way, virtually no time is spent for optimization or rollbacks at run time. Both set and bag semantics are considered.

## 1 Introduction

Correct and efficient integrity checking and maintenance are crucial issues in relational as well as deductive databases. Without proper devices that guarantee the consistency of data, the answers to queries cannot be trusted. Integrity constraints are logical formulas that characterize the consistent states of a database. They often require linear or worse complexity with respect to the size of the database for a complete check, which is too costly in any interesting case. To simplify a set of integrity constraints means to derive specialized checks that can be executed more efficiently at each update, employing the hypothesis that the data were consistent before the update itself. Ideally, these tests should be generated at database design time and executed before any update that may violate the integrity, so that expensive rollback operations become unneeded.

The principle of simplification of integrity constraints has been known and recognized for more than twenty years, with the first important results dating back to [23], and further developed by several other authors. However, the common practice in standard databases is still based on *ad hoc* techniques. Typically, database experts hand-code complicated tests in the application program producing the update requests or, alternatively, design triggers within the database management system that react upon certain update actions. Both methods are prone to errors and not particularly flexible with respect to changes in the schema or design of the database.

Automated simplification methods are therefore called for. Standard principles exist (e.g. *partial subsumption* [2]), but none of them seems to have emerged and gained ground in current database implementations, as all have significant limitations (see also section 4). In this paper we focus on one particular aspect of the simplification of integrity constraints (aggregates and arithmetic expressions), but the method we present, which extends [4], is fairly general, as summarized below.

- It uses a compiled approach: the integrity constraints are simplified at design-time and no computation is needed at run-time to optimize the check.
- It gives a pre-test: only consistency-preserving updates will eventually be given to the database and therefore no rollback operations are needed.
- The pre-test is a necessary and sufficient condition for consistency.
- The updates expressible in the language are so general as to encompass additions, deletions, changes, transactions and transactions patterns.
- It consists of transformation operators that can be reused for other purposes, such as data mining, data integration, abductive reasoning, etc.

The problem of the simplification of integrity constraints containing aggregates seems, with few rare exceptions, to have been largely ignored. In fact, the most comprehensive method we are aware of [10] can only produce post-tests that are sets of *instances* of the original integrity constraints and does not exploit the fact that those constraints are trusted in the database state before the update. We argue that, to be of any practical use, any simplification procedure must support aggregates and arithmetic constraints, which are among the most widespread constructs in current database technology.

In this paper we present a series of rewrite rules that allow to decompose aggregate expressions into simpler ones that can then be simplified by a constraint solver for arithmetic expressions or other rewrite rules. Such rules are almost directly translatable into an implementation and their practical significance is demonstrated with an extensive set of examples.

The paper is organized as follows. The simplification framework is presented in section 2, while its application to integrity constraints with aggregates and arithmetic built-ins is explained and exemplified in section 3. We review existing literature in the field in section 4 and provide concluding remarks in section 5.

## 2 A framework for simplification of integrity constraints

### 2.1 The language

We assume a function-free first-order typed language equipped with negation and built-ins for equality ( $\doteq$ ) and inequality ( $\neq$ ) of terms. The notions of (typed) *terms* ( $t, s, \dots$ ), *variables* ( $x, y, \dots$ ), *constants* ( $a, b, \dots$ ), *predicates* ( $p, q, \dots$ ), *atoms*, *literals* and *formulas* in general are defined as usual. The non-built-in predicates are called database predicates, and similarly for atoms and literals.

*Clauses* are written in the form  $Head \leftarrow Body$  where the head, if present, is an atom and the body a (perhaps empty) conjunction of literals. We assume in

this paper that the clauses are not recursive, i.e., the predicate used in the head of a clause does not occur in its body (directly in the same clause, or indirectly via other clauses). A *denial* is a headless clause and a *fact* is a bodiless clause; all other clauses are called *rules*. Logical equivalence between formulas is denoted by  $\equiv$ , entailment by  $\models$ , provability by  $\vdash$ . The notation  $\vec{t}$  indicates a sequence of terms  $t_1, \dots, t_n$  and the expressions  $p(\vec{t})$ ,  $\vec{s} \doteq \vec{t}$  and  $\vec{s} \neq \vec{t}$  are defined accordingly. We assume that all clauses are range restricted<sup>1</sup>, as defined below.

**Definition 1 (Range restriction).** *A variable is range bound if it occurs in a positive database literal or in an equality with a constant or another range bound variable. A conjunction of literals is range restricted if all variables in it are range bound. A clause is range restricted if its body is.*

Furthermore, the language allows built-in *arithmetic constraints* ( $<$ ,  $\leq$ ,  $>$ ,  $\geq$ ), whose arguments are arithmetic expressions. An *arithmetic expression* is a numeric term, an aggregate term, or a combination of arithmetic expressions via the four operations ( $+$ ,  $-$ ,  $\cdot$ ,  $/$ ), indicated in infix notation. An *aggregate term* is an expression of the form  $A_{[x]}(\exists x_1, \dots, x_n F)$ , where  $A$  is an aggregate operator,  $F$  is a disjunction of conjunctions of literals in which each disjunct is range restricted,  $x_1, \dots, x_n$  (the *local variables*) are some of the variables in  $F$ , and the optional  $x$ , if present, is the variable, called *aggregate variable*, among the  $x_1, \dots, x_n$  on which the aggregate function is calculated<sup>2</sup>. The non-local variables in  $F$  are called the *global variables*; if  $F$  has no global variables, then the argument of the aggregate, called the *key formula*, is said to be closed. The aggregate operators we consider are: Cnt (count), Sum, Max, Min, Avg (average).

*Example 1.* Given the person relation  $p(\text{name}, \text{age})$ ,  $\text{Avg}_y(\exists x, y p(x, y))$  is an aggregate term that indicates the average age of all persons, whereas  $\text{Cnt}(\exists x p(x, 30))$  the number of 30 years old persons.

Subsumption is a widely used syntactic principle that has the semantic property that the subsuming formula entails the subsumed one. We give here a definition for denials with arithmetic expressions.

**Definition 2 (Subsumption).** *Given two denials  $D_1, D_2$  with, resp., arithmetic constraints  $A_1, A_2$ ,  $D_1$  subsumes  $D_2$  (indicated  $D_1 \sqsubseteq D_2$ ) iff there is a substitution  $\sigma$  such that each database literal in  $D_1\sigma$  occurs in  $D_2$  and  $A_2 \vdash A_1\sigma$ .*

*Example 2.* Let  $\phi = \leftarrow \text{Cnt}(\exists x p(x, y)) < 10 \wedge q(y)$  and  $\psi = \leftarrow \text{Cnt}(\exists x p(x, b)) < 9 \wedge q(b) \wedge r(z)$ . Then  $\phi \sqsubseteq \psi$ .

In a database we distinguish three components: the *extensional database* (the facts), the *intensional database* (the rules or views) and the *constraint theory* (the integrity constraints) [15]. The constraint theory can be transformed in an equivalent one that does not contain any intensional predicates [2], and for ease of exposition we shall keep this assumption throughout the paper. For other cases

<sup>1</sup> Other texts use the terms “safe” and “allowed” to indicate the same notion.

<sup>2</sup> Alternatively, a number could be used instead of  $x$  to indicate the position of interest.

where intensional predicates might appear, we refer to the standard principle of *unfolding*, that consists in replacing such predicates by the extensional predicates they are defined upon. By *database state* we refer to the union of the extensional and the intensional parts only. As semantics of a database state  $D$ , with default negation for negative literals, we take its *standard model*, as  $D$  is here recursion-free and thus *stratified*. The truth value of a closed formula  $F$ , relative to  $D$ , is defined as its valuation in the standard model and denoted  $D(F)$ . (See, e.g., [24] for exact definitions for these and other common logical notions.)

**Definition 3 (Consistency).** *A database state  $D$  is consistent with a constraint theory  $\Gamma$  iff  $D(\Gamma) = \text{true}$ .*

The semantics of aggregates depends on the semantics underlying the representation of data. Two different semantics are of interest: set semantics, as traditionally used in logic, and bag semantics, typical of database systems. A state  $D$  contains, for every database predicate  $p$ , a relation  $p^D$ ; under set semantics  $p^D$  is a set of tuples, under bag semantic a bag (or multiset) of tuples, i.e., a set of  $\langle \text{tuple}, \text{multiplicity} \rangle$  pairs. Similarly, a range restricted closed formula  $F$  defines a new finite relation  $F^D$ , called its *extension*. Under set semantics the extension consists of all different answers that  $F$  produces over  $D$ ; under bag semantics the tuples are the same as for set semantics and the multiplicity of each tuple is the number of times it can be derived over  $D$ . We refer to [6, 8] for formal definitions. We note that Max and Min are indifferent of the semantics; for the other aggregates, we need only concentrate on bag semantics and introduce the notation  $\text{Cnt}_D$ ,  $\text{Sum}_D$ ,  $\text{Avg}_D$  when referring to the set of *distinct* tuples.

The method we describe here can handle general forms of update, including additions, deletions and changes. For any predicate  $p$  in the database, we can introduce rules that determine how the extension of  $p$  is augmented and reduced, respectively indicated with  $p^+$  and  $p^-$ , with the only restriction that  $p^+$  and  $p^-$  be disjoint and  $p^+$  range restricted.

**Definition 4 (Update).** *A deletion (resp. addition) referring to an extensional predicate  $p$  is a rule (resp. range restricted rule) whose head is an atom with a new name  $p^-$  (resp.  $p^+$ ) having the same arity as  $p$ . A non-empty bag of deletions and additions  $U$  is an update whenever, for every deletion  $p^-$  and addition  $p^+$  in the bag,  $(D \cup U)(\leftarrow p^+(\vec{x}) \wedge p^-(\vec{x})) = \text{true}$  for every state  $D$ , where  $\vec{x}$  is a sequence of distinct variables matching the arity of  $p$ .*

*For a state  $D$  and an update  $U$ , the updated state, indicated  $D^U$ , refers to the state obtained from  $D$  by changing the extension of every extensional predicate  $p$  to the extension of the formula  $(p(\vec{x}) \wedge \neg p^-(\vec{x})) \vee p^+(\vec{x})$  in  $(D \cup U)$ .*

Note that, according to definition 4, in both set and bag semantics, when an update  $U$  determines the deletion of a tuple  $p(\vec{c})$  from a relation  $p^D$  in a state  $D$ , then all of its occurrences in  $p^D$  are removed, i.e.,  $D^U(p(\vec{c})) = \text{false}$ . We also allow for special constants called *parameters* (written in boldface:  $\mathbf{a}$ ,  $\mathbf{b}$ , ...), i.e., placeholders for constants that permit to generalize updates into update *patterns*, which can be evaluated before knowing the actual values of the update itself. For example, the notation  $\{p^+(\mathbf{a}), q^-(\mathbf{a})\}$ , where  $\mathbf{a}$  is a parameter, refers

to the class of update transactions that add a tuple to the unary relation  $p$  and remove the same tuple from the unary relation  $q$ . We refer to [4] for precise definitions concerning parameters.

## 2.2 Semantic notions

We give now a characterization of simplification based on the notion of weakest precondition [9, 17], which is a semantic correctness criterion for a test to be run prior to the execution of the update, i.e., a test that can be checked in the present state but indicating properties of the prospective new state.

**Definition 5 (Weakest precondition).** *Let  $\Gamma$  and  $\Sigma$  be constraint theories and  $U$  an update.  $\Sigma$  is a weakest precondition of  $\Gamma$  with respect to  $U$  whenever  $D(\Sigma) \equiv D^U(\Gamma)$  for any database state  $D$ .*

The essence of simplification is the optimization of a weakest precondition based on the invariant that the constraint theory holds in the present state.

**Definition 6 (Conditional weakest precondition).** *Let  $\Gamma, \Delta$  be constraint theories and  $U$  an update. A constraint theory  $\Sigma$  is a  $\Delta$ -conditional weakest precondition ( $\Delta$ -CWP) of  $\Gamma$  with respect to  $U$  whenever  $D(\Sigma) \equiv D^U(\Gamma)$  for any database state  $D$  consistent with  $\Delta$ .*

Typically,  $\Delta$  will include  $\Gamma$ , but it may also contain other knowledge, such as further properties of the database that are trusted. The notion of CWP alone is not sufficient to fully characterize the principle of simplification. In [4, 5], an optimality criterion is introduced, which serves as an abstraction over actual computation times without introducing assumptions about any particular evaluation mechanism or referring to any specific database state. For reasons of space, we omit this discussion and approximate optimality with syntactic minimality (minimal number of literals), as shown in the following example.

*Example 3.* Consider the constraint theory  $\Gamma = \Delta = \{\leftarrow p(a) \wedge q(a), \leftarrow r(a)\}$  and the update  $U = \{p^+(a)\}$ . The semantically strongest, optimal and weakest  $\Delta$ -CWPs of  $\Gamma$  with respect to  $U$  are shown in the following table.

Strongest	Optimal	Weakest
$\{\leftarrow q(a), \leftarrow r(a)\}$	$\{\leftarrow q(a)\}$	$\{\leftarrow q(a) \wedge \neg p(a) \wedge \neg r(a)\}$

## 2.3 A simplification procedure without aggregates and arithmetic

We now describe the transformations that compose a simplification procedure for integrity constraints without aggregates and arithmetic built-ins that was introduced in [4]; we will extend it for these cases in section 3. Given an input constraint theory  $\Gamma$  and an update  $U$ , it returns a simplified theory to be tested before  $U$  is executed to guarantee that  $\Gamma$  will continue to hold after the update.

**Definition 7.** For a constraint theory  $\Gamma$  and an update  $U$ , the notation  $\text{After}^U(\Gamma)$  refers to a copy of  $\Gamma$  in which all occurrences of an atom of the form  $p(\vec{t})$  are simultaneously replaced by the unfolding of the formula:

$$(p(\vec{t}) \wedge \neg p^-(\vec{t})) \vee p^+(\vec{t}).$$

We assume that the result of the transformation of definition 7 is always given as a set of denials, which can be produced by using, e.g., De Morgan’s laws. The semantic correctness of **After** is expressed by the following property.

**Theorem 1.** For any update  $U$  and constraint theory  $\Gamma$ ,  $\text{After}^U(\Gamma)$  is a weakest precondition of  $\Gamma$  with respect to  $U$ .

Clearly, **After** is not in any “normalized” form, as it may contain redundant denials and sub-formulas (such as, e.g.,  $a \doteq a$ ). Moreover, the fact that the original integrity constraints hold in the current database state can be used to achieve further simplification. For this purpose, we define a transformation **Optimize** that exploits a given constraint theory  $\Delta$  consisting of trusted hypotheses to simplify the input theory  $\Gamma$ . The proposed implementation [22] is here described in terms of sound rewrite rules that tend towards a syntactically minimal constraint theory. In order to have termination, these rules are based on a straightforward provability principle based on procedures searching for specific patterns such as trivially satisfied (in)equalities and other purely syntactic notions, such as subsumption and a restricted application of resolution. **Optimize** removes from  $\Gamma$  every denial that is subsumed by another denial in  $\Delta$  or  $\Gamma$  and all literals that the (restricted) resolution principle proves to be redundant<sup>3</sup>.

**Definition 8.** Given two constraint theories  $\Delta$  and  $\Gamma$ ,  $\text{Optimize}_\Delta(\Gamma)$  is the result of applying the following rewrite rules on  $\Gamma$  as long as possible. In the following,  $x$  is a variable,  $t$  is a term,  $A, B$  are (possibly empty) conjunctions of literals,  $L$  is a literal,  $\sigma$  a substitution,  $\phi, \psi$  are denials,  $\Gamma'$  is a constraint theory.

$$\begin{aligned} \{\leftarrow x \doteq t \wedge A\} \cup \Gamma' &\Rightarrow \{\leftarrow A\{x/t\}\} \cup \Gamma' \\ \{\leftarrow A \wedge B\} \cup \Gamma' &\Rightarrow \{\leftarrow A\} \cup \Gamma' \text{ if } A \vdash B \\ \{\phi\} \cup \Gamma' &\Rightarrow \Gamma' \text{ if } \exists \psi \in (\Gamma' \cup \Delta) : \psi \sqsubseteq \phi \\ \{\leftarrow A\} \cup \Gamma' &\Rightarrow \Gamma' \text{ if } A \vdash \text{false} \\ \{(\leftarrow A \wedge \neg L \wedge B)\sigma\} \cup \Gamma' &\Rightarrow \{(\leftarrow A \wedge B)\sigma\} \cup \Gamma' \text{ if } (\Gamma \cup \Delta) \vdash \{\leftarrow A \wedge L\}^4 \end{aligned}$$

The operators defined so far can be assembled to define a procedure for simplification of integrity constraints, where the updates always take place from a consistent state. We also state its correctness and refer to [4] for a proof.

**Definition 9.** For a constraint theory  $\Gamma$  and an update  $U$ , we define

$$\text{Simp}^U(\Gamma) = \text{Optimize}_\Gamma(\text{After}^U(\Gamma)).$$

<sup>3</sup> Undecidability issues may arise, though, to prove that the resulting theory is indeed minimal; see [4] for discussion.

<sup>4</sup> The last rule can be implemented, e.g., with a data driven procedure that looks in the deductive closure of  $(\Gamma \cup \Delta)$  containing denials whose size is not bigger than the biggest denial in  $\Gamma$ .

**Theorem 2.** *Simp terminates on any input and, for any constraint theory  $\Gamma$  and update  $U$ ,  $\text{Simp}^U(\Gamma)$  is a  $\Gamma$ -CWP of  $\Gamma$  with respect to  $U$ .*

*Example 4.* Let  $\Gamma = \{\leftarrow p(x) \wedge q(x)\}$ ,  $U = \{p^+(\mathbf{a})\}$ . The simplification is as follows:  $\text{After}^U(\Gamma) = \{\leftarrow q(x) \wedge x \doteq \mathbf{a}, \leftarrow p(x) \wedge q(x)\}$ ,  $\text{Simp}^U(\Gamma) = \{\leftarrow q(\mathbf{a})\}$ .

### 3 Extension to aggregates and arithmetic

#### 3.1 Extension of the simplification framework

The constraint theory output by `After` holds in the current state if and only if the input theory holds in the updated state. Each predicate in the input theory is therefore replaced by an expression that indicates the extension it would have in the updated state. Unsurprisingly, theorem 1 can be proven also in the presence of aggregates and arithmetic, without modifying definition 7. We always express key formulas as disjunctions of conjunctions of literals; again, this can be done by simple (bag) semantics-preserving transformations such as De Morgan’s laws.

*Example 5.* Let  $\Gamma = \{\leftarrow \text{Cnt}(\exists x p(x)) < 10\}$ ,  $U = \{p^+(\mathbf{a})\}$ , then:

$$\text{After}^U(\Gamma) = \{\leftarrow \text{Cnt}(\exists x p(x) \vee x \doteq \mathbf{a}) < 10\}.$$

The new `Cnt` expression returned by `After` should indicate an increment by one with respect to the original expression. In order to determine this effect, we need to divide the expression into smaller pieces that can possibly be used during the simplification of weakest preconditions. To do that, we extend definition 8 with a set of sound rewrite rules for aggregates. Note that care must be taken when applying transformations that preserve logical equivalence on aggregates with bag semantics. Consider, for instance,  $\text{Sum}_x(\exists x p(x) \wedge x \doteq 10)$  and  $\text{Sum}_x(\exists x (p(x) \wedge x \doteq 1) \vee (p(x) \wedge x \doteq 1))$ . Their key formulas are logically equivalent, but in the latter, the tuple  $p(1)$  occurs twice as many times. In other words, the results may differ because the number of ways in which the key formula may succeed matters. With regard to arithmetic constraints and expressions, we base the applicability of these rules on the presence of a standard constraint solver for arithmetic; see, e.g., [7, 18]. To make the definitions more readable, we introduce *conditional expressions*, i.e., arithmetic expressions written `If  $C$  Then  $E_1$  Else  $E_2$` , which indicate arithmetic expression  $E_1$  if condition  $C$  holds,  $E_2$  otherwise; we also introduce the arithmetic operators `max` and `min` defined as conditional expressions. Furthermore, we take liberties in rewrite rules to omit portions of formulas and expressions with leading and trailing ellipses (...); they identify the same portions in both sides of the rules.

**Definition 10.** *For a constraint theory  $\Gamma$ ,  $\text{Optimize}(\Gamma)$  is the result of applying the following rewrite rules and those of definition 8 on  $\Gamma$  as long as possible, where  $x$  is a local variable,  $y$  a global one,  $\vec{x}$  a (possibly empty) sequence of distinct local variables  $x_1, \dots, x_n$  different from  $x$ ,  $A, B, C$  are range restricted conjunctions of literals ( $C$  with no local variables),  $E_1, E_2$  arithmetic expressions,  $t, s$  terms,  $c$  a constant,  $F$  a key formula,  $\text{Agg}$  any aggregate.*

### Rules for all aggregates

$$\begin{aligned}
\text{Agg}_{[x_i]}(\exists \vec{x} A \wedge t \doteq t) &\Rightarrow \text{Agg}_{[x_i]}(\exists \vec{x} A) \\
\text{Agg}_x(\exists \vec{x}, x A \wedge x \doteq x_i) &\Rightarrow \text{Agg}_{x_i}(\exists \vec{x} A\{x/x_i\}) \\
\text{Agg}_{[x_i]}(\exists \vec{x} A \wedge y \doteq t) &\Rightarrow \text{If } y \doteq t \text{ Then } \text{Agg}_{[x_i]}(\exists \vec{x} A\{y/t\}) \text{ Else } \perp^5 \\
\text{Agg}_{[x_i]}(\exists \vec{x} A) &\Rightarrow \perp \text{ if } A \vdash \text{false}
\end{aligned}$$

### Rules for Cnt

$$\begin{aligned}
\text{Cnt}(\exists \vec{x} A \vee B) &\Rightarrow \text{Cnt}(\exists \vec{x} A) + \text{Cnt}(\exists \vec{x} B) \\
\text{Cnt}(\exists \vec{x} A \wedge t \neq s) &\Rightarrow \text{Cnt}(\exists \vec{x} A) - \text{Cnt}(\exists \vec{x} A \wedge t \doteq s) \\
\text{Cnt}(\exists \vec{x}, x A \wedge x \doteq t) &\Rightarrow \text{Cnt}(\exists \vec{x} A\{x/t\})^6 \\
\text{Cnt}(\text{true}) &\Rightarrow 1
\end{aligned}$$

### Rules for Sum and Avg

$$\begin{aligned}
\text{Sum}_{x_i}(\exists \vec{x} A \vee B) &\Rightarrow \text{Sum}_{x_i}(\exists \vec{x} A) + \text{Sum}_{x_i}(\exists \vec{x} B) \\
\text{Sum}_{x_i}(\exists \vec{x} A \wedge t \neq s) &\Rightarrow \text{Sum}_{x_i}(\exists \vec{x} A) - \text{Sum}_{x_i}(\exists \vec{x} A \wedge t \doteq s) \\
\text{Sum}_{x_i}(\exists \vec{x}, x A \wedge x \doteq t) &\Rightarrow \text{Sum}_{x_i}(\exists \vec{x} A\{x/t\}) \\
\text{Sum}_x(\exists \vec{x}, x A \wedge x \doteq c) &\Rightarrow c \cdot \text{Cnt}(\exists \vec{x} A\{x/c\}) \\
\text{Avg}_x(F) &\Rightarrow \text{Sum}_x(F)/\text{Cnt}(F)
\end{aligned}$$

### Rules for Max (and, symmetrically, for Min)

$$\begin{aligned}
\text{Max}_{x_i}(\exists \vec{x} A \vee B) &\Rightarrow \max(\text{Max}_{x_i}(\exists \vec{x} A), \text{Max}_{x_i}(\exists \vec{x} B)) \\
\text{Max}_{x_i}(\exists \vec{x}, x A \wedge x \doteq t) &\Rightarrow \text{Max}_{x_i}(\exists \vec{x} A\{x/t\}) \\
\text{Max}_x(\exists \vec{x}, x A \wedge x \doteq c) &\Rightarrow \text{If } \exists \vec{x} A\{x/c\} \text{ Then } c \text{ Else } -\infty
\end{aligned}$$

### Rules for Cnt<sub>D</sub> that differ from Cnt<sup>7</sup>

$$\begin{aligned}
\text{Cnt}_D(\exists \vec{x} A \vee B) &\Rightarrow \text{Cnt}_D(\exists \vec{x} A) + \text{Cnt}(\exists \vec{x} B) - \text{Cnt}(\exists \vec{x} A \wedge B) \\
\text{Cnt}_D(C) &\Rightarrow \text{If } C \text{ Then } 1 \text{ Else } 0
\end{aligned}$$

### Rules for conditional expressions

$$\{\leftarrow \dots \text{If } C \text{ Then } E_1 \text{ Else } E_2 \dots\} \Rightarrow \{\leftarrow C \wedge \dots E_1 \dots, \leftarrow \neg C \wedge \dots E_2 \dots\}$$

Definition 9 (Simp) refers now to the Optimize operator of definition 10. Theorem 2 still holds as long as we assume a terminating solver for arithmetic [18].

## 3.2 Examples of simplification

We show now a series of examples that demonstrate the behavior of the rules and the simplification procedure in various cases; for readability, we leave out some of the trivial steps. When it is clear from the context that a variable is local, we omit the indication of its existential quantifier.

<sup>5</sup> Provided that  $t$  is not a local variable. The  $\perp$  symbol indicates the value that applies to an empty bag of tuples (0 for Cnt, Sum,  $-\infty$  for Max,  $+\infty$  for Min, etc.).

<sup>6</sup> In all rules with substitutions, the replacing term or constant can also be an arithmetic expression as long as the substitution does not insert it in a database literal.

<sup>7</sup> Similar rules can be written for Sum<sub>D</sub> and Avg<sub>D</sub> referring to Sum and Avg.

*Example 6.* Consider the aggregates  $A_1 = \text{Cnt}(\exists x p(x) \wedge x \neq \mathbf{a})$  and  $A_2 = \text{Sum}_x(\exists x p(x) \wedge x \neq \mathbf{a})$ ,  $\mathbf{a}$  a numeric parameter. The following rewrites apply:

$$\begin{aligned} A_1 &\Rightarrow \text{Cnt}(\exists x p(x)) - \text{Cnt}(\exists x p(x) \wedge x \doteq \mathbf{a}) \Rightarrow \text{Cnt}(\exists x p(x)) - \text{Cnt}(p(\mathbf{a})). \\ A_2 &\Rightarrow \text{Sum}_x(p(x)) - \text{Sum}_x(p(x) \wedge x \doteq \mathbf{a}) \Rightarrow \text{Sum}_x(p(x)) - \mathbf{a} \cdot \text{Cnt}(p(\mathbf{a})). \end{aligned}$$

*Example 7 (5 continued).* Let  $\Gamma = \{\leftarrow \text{Cnt}(\exists x p(x)) < 10\}$ ,  $U = \{p^+(\mathbf{a})\}$ , then:

$$\begin{aligned} \text{Simp}^U(\Gamma) &= \text{Optimize}_\Gamma(\{\leftarrow \text{Cnt}(\exists x p(x) \vee x \doteq \mathbf{a}) < 10\}) = \\ &= \text{Optimize}_\Gamma(\{\leftarrow \text{Cnt}(\exists x p(x)) + \text{Cnt}(\exists x x \doteq \mathbf{a}) < 10\}) = \\ &= \text{Optimize}_\Gamma(\{\leftarrow \text{Cnt}(\exists x p(x)) + 1 < 10\}) = \text{true}. \end{aligned}$$

The update increments the count of  $p$ -tuples, which was known ( $\Gamma$ ) to be at least 10 before the update, so this increment cannot undermine the validity of  $\Gamma$  itself. The last step, obtained via subsumption, allows to conclude that no check is necessary to guarantee the consistency of the updated database state.

*Example 8.* Let  $\Gamma = \{\leftarrow \text{Cnt}_b(\exists x p(x)) \neq 10\}$  (there must be exactly 10 *distinct*  $p$ -tuples) and  $U = \{p^+(\mathbf{a})\}$ . With a set semantics, the increment of the count depends on the existence of the tuple  $p(\mathbf{a})$  in the state:

$$\begin{aligned} \text{Simp}^U(\Gamma) &= \text{Optimize}_\Gamma(\{\leftarrow \text{Cnt}_b(\exists x p(x) \vee x \doteq \mathbf{a}) \neq 10\}) = \\ &= \text{Optimize}_\Gamma(\{\leftarrow \text{Cnt}_b(\exists x p(x)) + 1 - \text{Cnt}_b(p(\mathbf{a})) \neq 10\}) = \\ &= \text{Optimize}_\Gamma(\{\leftarrow \text{Cnt}_b(\exists x p(x)) + 1 - \text{If } p(\mathbf{a}) \text{ Then } 1 \text{ Else } 0 \neq 10\}) = \\ &= \text{Optimize}_\Gamma(\{\leftarrow p(\mathbf{a}) \wedge \text{Cnt}_b(\exists x p(x)) + 1 - 1 \neq 10, \\ &\quad \leftarrow \neg p(\mathbf{a}) \wedge \text{Cnt}_b(\exists x p(x)) + 1 - 0 \neq 10\}) = \\ &= \{\leftarrow \neg p(\mathbf{a})\}. \end{aligned}$$

The arithmetic constraint solver intervenes in the last step using the knowledge from  $\Gamma$  that the original  $\text{Cnt}_b$  expression is equal to 10.

*Example 9.* When global variables occur, conditional expressions are used to separate different cases. Let  $\Gamma = \{\leftarrow \text{Cnt}(\exists x p(x, y)) > 10 \wedge q(y)\}$  (there cannot be more than 10  $p$ -tuples whose second argument is in  $q$ ) and  $U = \{p^+(\mathbf{a}, \mathbf{b})\}$ .

$$\begin{aligned} \text{Simp}^U(\Gamma) &= \text{Optimize}_\Gamma(\text{After}^U(\Gamma)) = \\ &= \text{Optimize}_\Gamma(\{\leftarrow \text{Cnt}(\exists x p(x, y)) + \text{Cnt}(\exists x x \doteq \mathbf{a} \wedge y \doteq \mathbf{b}) > 10 \wedge q(y)\}) = \\ &= \text{Optimize}_\Gamma(\{\leftarrow \text{Cnt}(\exists x p(x, y)) + \text{Cnt}(y \doteq \mathbf{b}) > 10 \wedge q(y)\}) = \\ &= \text{Optimize}_\Gamma(\{\leftarrow \text{Cnt}(\exists x p(x, y)) + \text{If } y \doteq \mathbf{b} \text{ Then } 1 \text{ Else } 0 > 10 \wedge q(y)\}) = \\ &= \text{Optimize}_\Gamma(\{\leftarrow y \doteq \mathbf{b} \wedge \text{Cnt}(\exists x p(x, y)) + 1 > 10 \wedge q(y), \\ &\quad \leftarrow y \neq \mathbf{b} \wedge \text{Cnt}(\exists x p(x, y)) + 0 > 10 \wedge q(y)\}) = \\ &= \{\leftarrow \text{Cnt}(\exists x p(x, \mathbf{b})) > 9 \wedge q(\mathbf{b})\}. \end{aligned}$$

In the last step, the second constraint is subsumed by  $\Gamma$  and thus eliminated.

*Example 10.* We propose now an example with a complex update. Let  $e(x, y, z)$  represent employees of a company, where  $x$  is the name,  $y$  the years of service and  $z$  the salary. The company's policy is expressed by

$$\Gamma = \{\leftarrow e(x, y, z) \wedge z \doteq \text{Max}_{z_l}(e(x_l, y_l, z_l)) \wedge (5 > y \vee y > 8)\}^8$$

<sup>8</sup> For compactness, we use  $\vee$  instead of writing two different denials.

i.e., the seniority of the best paid employee must be between 5 and 8 years, and

$$U = \{e^+(x, y_2, z) \leftarrow e(x, y_1, z) \wedge y_2 \doteq y_1 + 1, e^-(x, y, z) \leftarrow e(x, y, z)\}$$

is the update transaction that is executed to increase the seniority of all employees at the end of the year. Note that the application of  $\text{After}^U$  to a literal of the form  $e(x, y, z)$  generates  $(e(x, y, z) \wedge \neg e(x, y, z)) \vee (e(x, y', z) \wedge y \doteq y' + 1)$  in which the first disjunct is logically equivalent to *false* and can, thus, be removed. The aggregate in  $\text{After}^U(\Gamma)$  comes to coincide, modulo renaming, with the original one in  $\Gamma$ :  $\text{Max}_{z_l}(e(x_l, y'_l, z_l) \wedge y_l \doteq y'_l + 1) \Rightarrow \text{Max}_{z_l}(e(x_l, y'_l, z_l))$ . We have:

$$\begin{aligned} \text{Optimize}_\Gamma(\{\leftarrow e(x, y', z) \wedge y \doteq y' + 1 \wedge z \doteq \text{Max}_{z_l}(e(x_l, y_l, z_l)) \wedge (5 > y \vee y > 8)\}) = \\ = \{\leftarrow e(x, y', z) \wedge z \doteq \text{Max}_{z_l}(e(x_l, y_l, z_l)) \wedge y' > 7\} = \text{Simp}^U(\Gamma). \end{aligned}$$

## 4 Related works

As pointed out in section 1, the principle of simplification of integrity constraints is essential for optimizations in database consistency checking. A central quality of any good approach to integrity checking is the ability to verify the consistency of a database to be updated *before* the execution of the transaction in question, so that inconsistent states are completely avoided. Several approaches to simplification do not comply with this requirement, e.g., [23, 21, 11, 14]. Among these, we mention the resolution-based principle of partial subsumption [2, 14], which is important in semantic query optimization, although its use for simplification of integrity constraints is rather limited, as transactions are only partly allowed. Other methods, e.g., [16], provide pre-tests that, however, are not proven to be necessary conditions; in other words, if the tests fail, nothing can be concluded about the consistency. Qian presented in [25] a powerful simplification procedure that, however, does not allow more than one update action in a transaction to operate on the same relation and has no mechanism corresponding to parameters, thus requiring to execute the procedure for each update. Most works in the field lack a characterization of what it means for a formula to be simplified. Our view is that a transformed integrity constraint, in order to qualify as “simplified”, must represent a minimum (or a good approximation thereof) in some ordering that reflects the effort of actually evaluating it. We propose a simple ordering based on the number of literals but we have no proof that our algorithm reaches a minimum in all cases. Most simplification methods do not allow recursion. The rare exceptions we are aware of (e.g., [21, 20, 3]) hardly provide useful simplified checks: when recursive rules are present, these methods typically output the same constraints as in the input set. None of the methods described so far can handle aggregates and only [3] considers arithmetic constraints.

The constraint solver for finite domains described in [7] and available in current Prolog systems is able to handle the arithmetic part of most of the examples and rules described in this paper (for integers). For a survey on constraint solvers and constraint logic programming we refer to [18].

In [10] Das extends the simplification method of [21] and applies it to aggregates. However, the hypotheses about the consistency of the database prior to the update is not exploited; consequently, the simplified test can only be a set of instances of the original constraints. In our example 7, for instance, Das' method would return the initial constraint theory, whereas we were able to conclude that no check was needed. In [8], the authors describe query optimization techniques and complexity results under set and bag semantics and introduce the important *bag-set semantics*, i.e., the semantics corresponding to the assumption that database relations are sets, but queries may generate duplicates due to projection. This semantics has been used in subsequent work, e.g., [6], to approach problems concerning queries with aggregates, such as query rewriting using views. A definition of the semantics of SQL with aggregates is given in [1], where it is shown how to translate a subset of SQL into relational calculus and algebra and general strategies for query optimization are investigated for such cases. Further investigation on the semantics of aggregates is given in [19, 12, 27]. In [13] it is shown how to optimize, by *propagation*, queries with maximum and minimum aggregates in a possibly recursive setting. User-defined and on-line aggregates are described in [26] and their use is demonstrated for data mining and other advanced database applications.

## 5 Conclusion

We presented a set of rewrite rules that can be applied to a set of integrity constraints containing aggregates and arithmetic expressions in order to obtain simplified tests that serve to ensure the consistency of the database before any update is made. Our approach is a first attempt to simplify aggregate expressions in a systematic way producing a necessary and sufficient condition for consistency. The rules we have presented are of practical relevance, as shown in a number of examples, and should be considered as a starting point for possibly more complete and refined simplification procedures. Although not all details were spelled out, we have shown the interaction with a constraint solver to handle combinations of arithmetic constraints. Future directions include, to name a few, the extension to aggregates with recursion and user-defined aggregates.

**Acknowledgements** The author wishes to thank Henning Christiansen and Amos Scisci for many helpful comments and suggestions.

## References

1. Bülzingsloewen, G. Translating and Optimizing SQL Queries Having Aggregates. *VLDB 1987*: 235–243, Morgan Kaufmann (1987).
2. Chakravarthy, U. S., Grant, J., Minker, J. Foundations of semantic query optimization for deductive databases. *Foundations of Deductive Databases and Logic Programming*: 243–273, Minker, J. (Ed.), Morgan Kaufmann (1988).
3. Chakravarthy, U., Grant, J., Minker, J. Logic-based approach to semantic query optimization. *ACM TODS* 15(2): 162–207, ACM Press (1990).

4. Christiansen, H., Martinenghi, D., Simplification of database integrity constraints revisited: A transformational approach. Presented at *LOPSTR 2003*. Available at <http://www.dat.ruc.dk/~henning/LOPSTR03.pdf> (2003).
5. Christiansen, H., Martinenghi, D. Simplification of integrity constraints for data integration. *FoIKS 2004*, to appear in LNCS, Springer (2004).
6. Cohen, S., Nutt, W., Serebrenik, A. Algorithms for Rewriting Aggregate Queries Using Views. *ADBIS-DASFAA 2000*: 65–78, LNCS 1884, Springer (2000).
7. Carlsson, M., Ottosson, G., Carlson, B. An Open-Ended Finite Domain Constraint Solver. *PLILP 1997*: 191–206, LNCS 1292, Springer (1997).
8. Chaudhuri, S., Vardi, M. Optimization of *real* conjunctive queries. *ACM SIGACT-SIGMOD-SIGART PODS 1993*: 59–70, ACM Press (1993).
9. Dijkstra, E.W. A Discipline of Programming. Prentice-Hall (1976).
10. Das, S. Deductive Databases and Logic Programming. Addison-Wesley (1992).
11. Decker, H., Celma, M. A slick procedure for integrity checking in deductive databases. *ICLP 1994*: 456–469, MIT Press (1994).
12. Denecker, M., Pelov, N., Bruynooghe, M. Ultimate Well-founded and Stable Semantics for Logic Programs with Aggregates. *ICLP 2001*: 212–226, LNCS 2237, Springer (2001).
13. Furfaro, F., Greco, S., Ganguly, S., Zaniolo, C. Pushing extrema aggregates to optimize logic queries. *Inf. Syst.* 27(5): 321–343, Elsevier (2002).
14. Grant, J., Minker, J. Integrity Constraints in Knowledge Based Systems. In *Knowledge Eng. II, Applications*: 1–25, Adeli, H. (Ed.), McGraw-Hill (1990).
15. Godfrey, P., Grant, J., Gryz, J., Minker, J. Integrity Constraints: Semantics and Applications. *Logics for Databases and Information System*: 265–306, Chomicki, J. Saake, G. (Eds.), Kluwer (1988).
16. Henschen, L., McCune, W., Naqvi, S. Compiling Constraint-Checking Programs from First-Order Formulas. *Advances in Database Theory 1982*, volume 2: 145–169, Gallaire, H., Nicolas, J.-M., Minker, J. (Eds.), Plenum Press (1984).
17. Hoare, C.A.R. (1969). An axiomatic basis for computer programming. *Commun. ACM* 12(10): 576–580, ACM Press (1969).
18. Jaffar, J., Maher, M. Constraint Logic Programming: A Survey. *JLP* 19/20: 503–581 (1994).
19. Kemp, D., Stuckey, P. Semantics of Logic Programs with Aggregates. *ISLP 1991*: 387–401, MIT Press (1991).
20. Leuschel, M., De Schreye, D. Creating Specialised Integrity Checks Through Partial Evaluation of Meta-Interpreters. *JLP* 36(2): 149–193 (1998).
21. Lloyd, J., Sonenberg, L., Topor, R. (1987). Integrity Constraint Checking in Stratified Databases. *JLP* 4(4): 331–343 (1987).
22. Martinenghi, D. A Simplification Procedure for Integrity Constraints. *World Wide Web*, <http://www.dat.ruc.dk/~dm/spic/index.html> (2003).
23. Nicolas, J.-M. Logic for Improving Integrity Checking in Relational Data Bases., *Acta Inf.* 18: 227–253 (1982).
24. Nilsson, U., Małczyński, J. Logic, Programming and Prolog (2nd ed.), John Wiley & Sons Ltd. (1995).
25. Qian, X. An Effective Method for Integrity Constraint Simplification. *ICDE 1988*: 338–345, IEEE Computer Society (1988).
26. Wang, H., Zaniolo, C. User-Defined Aggregates in Database Languages, *DBPL 1999*: 43–60, LNCS 1949, Springer (2002).
27. Zaniolo, C. Key Constraints and Monotonic Aggregates in Deductive Databases. *Comput. Logic*: 109–134, LNAI 2408, Springer (2002).